Fine-Grained Disclosure of Access Policies

Claudio A. Ardagna[1], Sabrina De Capitani di Vimercati[1], Sara Foresti[1],
Gregory Neven[2], Stefano Paraboschi[3], Franz-Stefan Preiss[2],
Pierangela Samarati[1], and Mario Verdicchio[3]

[1] Università degli Studi di Milano, 26013 Crema, Italy
{firstname.lastname}@unimi.it
[2] IBM Research Zürich, Rüschlikon, Switzerland
{nev,frp}@zurich.ibm.com
[3] Università degli Studi di Bergamo, 24044 Dalmine, Italy
{parabosc,mario.verdicchio}@unibg.it

Abstract. In open scenarios, where servers may receive requests to access their services from possibly unknown clients, access control is typically based on the evaluation of (certified or uncertified) properties, that clients can present. Since assuming the client to know a-priori the properties she should present to acquire access is clearly limiting, servers should be able to respond to client requests with information on the access control policies regulating access to the requested services. In this paper, we present a simple, yet flexible and expressive, approach for allowing servers to specify *disclosure policies*, regulating if and how access control policies on services can be communicated to clients. Our approach allows fine-grain specifications, thus capturing different ways in which policies, and portions thereof, can be communicated. We also define properties that can characterize the client view of the access control policy.

1 Introduction

Despite the great improvements of ICT systems in access to information and communication, there are important user- and server-side requirements that are currently not completely satisfied. On one hand, users want to access resources without having to deal with the creation of accounts, the explicit memorization of passwords, or the disclosure of sensitive personal information. On the other hand, service providers need robust and effective ways to identify users, and to grant access to resources only to those users satisfying given conditions. User-side credentials have a great potential to satisfy these requirements. Using credentials, users are freed from the burden of having to keep track of a multitude of accounts and passwords. With credential-based, or more generically, attribute-based access control policies, servers can regulate access to their services based on properties and certificates that clients can present. Such a scenario changes how the access control works, not requesting servers to evaluate the policies with complete knowledge of clients and their properties, but rather to communicate to clients the policies that they should satisfy to have their requests possibly permitted. This aspect has been under the attention of the research and development communities for more than a decade and several solutions have been proposed addressing different issues [6, 13–15].

Although the need for supporting attribute-based access control has been recognized, current emerging practical access control solutions [9] still lack the ability of working with unknown clients and assume a-priori knowledge of their credentials and properties. However, requiring the client to know a-priori which information she needs to present takes away most of the benefits of supporting attribute-based access control and clearly limits its applicability. The communication to the client of the policy regulating access to a service is not straightforward. For instance, consider a policy restricting access to a service only to people with US nationality. How should a server communicate such a policy? Should the server present it completely? Or should it just ask the client to state her nationality? Clearly, there is not a unique response, and which option is to be preferred may depend on the context and on the information involved. We note that communicating the complete policy favors the privacy of the client (since she can avoid disclosing her properties if they would not satisfy the conditions in the policy). By contrast, communicating only the attributes needed for the evaluation of the policy favors the privacy of the server (since the specific conditions in its policy are not disclosed). With respect to the server, different portions of a policy might have different confidentiality requirements. For instance, consider a service open to all people older than *18* and not working for a company blacklisted at the server. When communicating to the client that she has to disclose her age and company before accessing the service, the server might not mind communicating that the service is restricted only to people who are older than *18*, but might at the same time not want to disclose that the client will be checked against a blacklist that the server is maintaining.

In this paper, we provide a simple, yet expressive and flexible, approach for enabling servers to specify, when defining their access control policies, if and how the policy should be communicated to the client. Our approach applies to generic attribute-based access control policies and is therefore compatible with different languages, including logic-based approaches [5] as well as the established XACML standard [9]. The contributions of this paper can be summarized as follows. First (Sections 2-3), exploiting a graphical representation of the access control policies in terms of their expression tree, we define disclosure policies as colors (green, yellow, or red) that can be applied to the different nodes in the tree, ensuring a fine-grained support and providing expressiveness and flexibility in establishing disclosure regulations. Second (Section 4), we illustrate how disclosure policies specified by the given coloring are enforced to determine how the policy should be communicated to the client. Third (Section 5), we identify different properties of the client policy view. Fourth (Section 6), we briefly illustrate how our approach can provide a support for credential ontologies and for emerging solutions allowing clients to present a proof of the satisfiability of given conditions without revealing their properties. Finally, we discuss related work (Section 7) and give our conclusions (Section 8).

2 Preliminary Concepts

We consider a scenario where clients and servers may interact without having a-priori knowledge of each other. Servers offer services/resources whose access is regulated by access control policies. Clients have a *portfolio* of properties (called attributes) that they enjoy. The client portfolio includes both attributes in certificates signed by a third party (credentials) and attributes declared by the client herself (declarations) [4]. We assume that each credential is characterized by attribute `type` that identifies the attributes that the credential certifies.

Access control policies at the server side specify the attributes that a client should have to gain access to the services. The policy applicable to a given service can then be seen as a boolean formula over basic conditions of the form $(p, term_1, \ldots, term_n)$, where p is a predicate operator and $term_i, i = 1, \ldots, n$, are attributes (e.g., `country`, `birth_date`) or constant values (e.g., *USA*, *1970/04/12*) corresponding to the operands of the predicate operator. While our approach can be applied to generic predicates with an arbitrary number of operands, in the remainder of the paper for concreteness and simplicity, we consider binary predicate operators, where conditions are expressed according to the classical infix notation $(term_1\ p\ term_2)$. A server may specify whether the attributes appearing in basic conditions must be certified or can be declared by a client. Requests for a certified attribute `a` are expressed as c.`a`, where c is a symbol denoting a credential. In addition, if attribute `a` must be certified by a credential of a given type t, a basic condition c.`type`$=t$ must be specified in the policy. Note that the same credential symbol may appear in multiple conditions. In this case, such conditions refer to, and must be satisfied by, the same credential. By contrast, if a different symbol is used, the corresponding conditions can be satisfied by different credentials. For instance, policy c_1.`type`=*Passport* $\wedge$ c_1.`country`=*USA* $\wedge$ c_2.`type`=*CreditCard* $\wedge$ c_2.`expiration`>*2010/12/31* states that access is allowed if a client presents a passport proving US citizenship and a credit card with expiration date greater than *2010/12/31*.

A policy can be represented as a boolean expression tree as follows.

Definition 1 (Policy tree). *Let P be a boolean formula representing a policy. A* policy tree $T(N)$ *representing P is a tree where:*

- *there is a node $n \in N$ for each operator, attribute, and value appearing in P;*
- *each node $n \in N$ has a label, denoted* label(n)*, corresponding to the operator, attribute, or value represented by n;*
- *each node $n \in N$ representing an operator has an ordered set of children, denoted* children(n)*, corresponding to the roots of the subtrees representing the operands of* label(n).

Note that operators are internal nodes, and attributes and values are leaf nodes. In the graphical representation of the policy tree, we distinguish nodes representing constant values (in contrast to attributes or operators) by representing them with a square. Figure 1(a) illustrates the policy tree representing

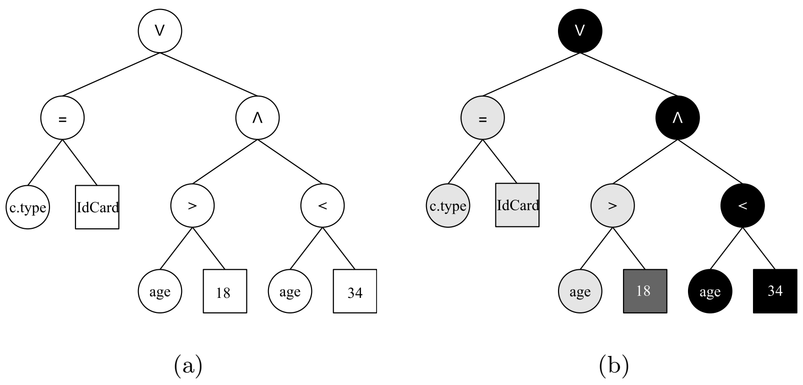

Fig. 1. An example of policy tree (a) and of colored policy tree (b)

a policy stating that access is allowed if the client either *i)* presents an identity card, or *ii)* discloses her age and the age is between 18 and 34. Formally: $(c.\texttt{type}{=}\mathit{IdCard})\vee(\texttt{age}{>}\mathit{18}\wedge\texttt{age}{<}\mathit{34})$.

3 Disclosure Policy Specifications

The main goal of this paper is to provide the server with a means to regulate how its access control policies should be communicated to clients. In fact, the server might consider its access control policy, or part of it, as confidential. To provide maximum flexibility and expressiveness, we define a fine-grained approach where each term and predicate operator appearing in a condition, as well as each boolean operator combining different conditions, can be subject to a *disclosure policy* that regulates how the term, predicate operator, or boolean operator should be protected and then communicated to the client. In other words, each node in the policy tree can be associated with a disclosure policy regulating if and how the existence of the node and its label should be visible to the client. With respect to the label, a disclosure policy can state whether the label of a node can be disclosed. With respect to the existence of a node and therefore the structure of the tree, a disclosure policy can state whether the structure has to be preserved or can be possibly obfuscated by removing nodes from the tree.

The disclosure policy associated with each node in the tree is expressed as a color (green, yellow, or red), which in the figures in the paper corresponds to the light gray (green), dark grey (yellow), and black (red). The specification of disclosure policies results in a *colored policy tree*, formally defined as follows.

Definition 2 (Colored policy tree). *A* colored policy tree $T^{\lambda}(N,\lambda)$ *is a policy tree* $T(N)$ *with a coloring function* $\lambda{:}N{\rightarrow}\{\text{green,yellow,red}\}$ *that maps each node in* N *onto a color in* $\{\text{green,yellow,red}\}$.

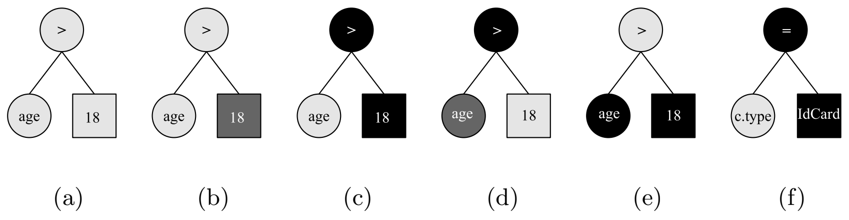

Fig. 2. Examples of disclosure policies (a)-(c) and of non well defined colorings (d)-(f)

Figure 1(b) illustrates a possible coloring for the policy tree in Figure 1(a). The semantics of the different colors, with respect to the release to the client of the colored node, is as follows.

- *Green.* A green node is released as it is. For instance, consider the policy tree in Figure 2(a). Since all the nodes in the policy tree are green, the policy is disclosed as it is, that is, (`age`>*18*).
- *Yellow.* A yellow node is obfuscated by only removing its label; the presence of the node as well as of its children is preserved. For instance, consider the policy tree in Figure 2(b). The disclosed policy is (`age`>_), communicating to the client that only clients having `age` greater than a threshold can access the resource, without disclosing the threshold applied.
- *Red.* A red node is obfuscated by removing its label and possibly its presence in the tree. For instance, consider the policy tree in Figure 2(c). The disclosed policy is composed of attribute `age` and only reveals to the client that the policy evaluates attribute `age`, without disclosing the condition on it.

Note that, while the server has the ability and flexibility of coloring each and every node in the policy tree, of course a default policy could be applied, setting to a predefined color all nodes of a tree (or sub-trees) unless differently specified. Note also the two extremes of the default coloring, corresponding to having the policy tree all green (the policy is fully disclosed) or all red (nothing is disclosed). Also, although in principle a node in a policy tree can be arbitrarily colored (i.e., any disclosure policy can be associated with any node in a tree), not all the colorings of a policy tree are *well defined*. A coloring function is well defined if all the following conditions are satisfied.

- For each green leaf representing a constant value, its sibling representing an attribute is green, and its parent, representing a predicate operator, is not red. The reason for this constraint is to not allow cases where the only information releasable to the client is the constant value against which an attribute is compared, without releasing neither the attribute nor the predicate operator. Figure 2(d) reports an example of coloring violating this constraint.

– Each green node representing a predicate operator must have at least a non red child. The reason for this constraint is analogous to the one above. In fact, releasing a predicate operator (e.g., $>$, $<$, $=$) without its operands would be meaningless. Figure 2(e) illustrates an example of coloring violating this constraint.
– For each subtree representing a basic condition on attribute `type`, the nodes in the subtree are either all green or all red. The reason for this constraint is to ensure that if the information that there is a condition on the type of credential in the policy is released to the client, also the specific type of credential is disclosed. In fact, it would be meaningless to state that only credentials of a given type are accepted, without disclosing the type. Figure 2(f) illustrates an example of policy violating this constraint.

Apart from the constraints above restricting the diversity of colors within basic conditions of the policy, any color can be assigned to the different nodes of a policy tree, each producing a different way in which the server may wish to communicate its policy to the client. In other words, each coloring produces a possible view of the client on the policy tree.

4 The Transformation Process: Client Policy View

The colored policy tree T^λ must be transformed into an equivalent *client policy tree view* T', which is communicated to the client. T' is obtained by applying transformation rules that: *i)* remove the label of yellow and red nodes, *ii)* remove unnecessary red leaves, and *iii)* collapse red internal nodes in a parent-child relationship into a single red node. These transformation rules[4] are illustrated in Figure 3, where white nodes are nodes of any color (i.e., either green, yellow, or red) and leaf nodes have two nil nodes represented by small black squares. The transformation rules are classified in three categories, *prune*, *collapse*, and *hide label*, depending on the role they play in the transformation process, as described in the following.

Prune. Prune rules operate on internal nodes whose children are leaf nodes. These rules are intended to remove unnecessary red leaves, if any. Given a subtree rooted at node n, two kinds of prune rules may apply.

– *Red predicate*. If node n is red and *label*(n) is a predicate operator, its red child (if any) is removed from the tree. We note that n must have either a green child, representing an attribute, or a yellow child (otherwise the coloring function λ characterizing T^λ would not be well defined). For instance, consider condition (`age`$>$*18*) and suppose that both node $>$ and node *18* are red. The obfuscated condition (`age` _ _) and the release of attribute `age` are equivalent with respect to the information disclosed to a client.

[4] We report the rules where the right child is red. The case where the left child is red is symmetric.

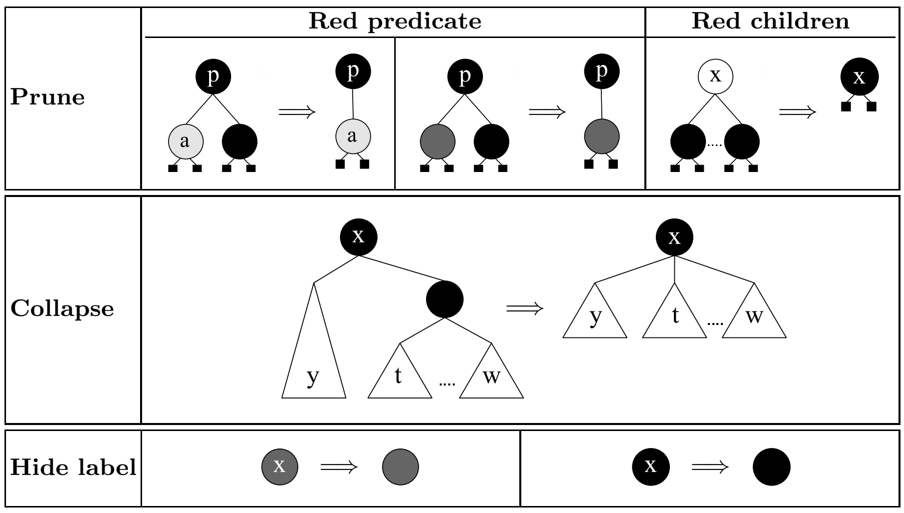

Fig. 3. Prune, collapse, and hide label rules

– *Red children.* If all the children of n are red, they are removed from the tree and $\lambda(n)$ is set to red, since the release of the operator represented by n without its operands is meaningless. For instance, consider condition (`age`>*18*) and suppose that `age` and *18* are red and > is yellow. The release of the obfuscated condition (_ _ _) is meaningless for a client and can then be considered equivalent to the release of one leaf red node.

Collapse. The collapse rule operates on internal red nodes and removes their non-leaf red children, if any. Given a subtree rooted at a red node n with a red non-leaf child n', node n' is removed from the tree and its children replace n' in the ordered list of the children of n. This rule, recursively applied, collapses all red nodes forming a path in a single red node, since their labels have been suppressed and their presence in the policy tree can be obfuscated. For instance, consider the policy tree in Figure 1(a) and suppose that nodes $\vee$ and $\wedge$ are red. The release of the obfuscated condition (`c.type`=*IdCard*) _ ((`age`>*18*) _ (`age`<*34*)) does not provide any additional information to a client with respect to the separate release of each condition.

Hide label. Hide label rules operate on yellow and red nodes by removing their labels.

The transformation process works by traversing the tree with a post-order visit and, at each node, applies the rules in the order prune, collapse, and hide. Figure 4 illustrates the pseudocode of the transformation algorithm that takes a colored policy tree T^λ as input and returns the corresponding policy tree view T'. The algorithm first initializes T^λ to T', by assigning N' and λ' to N and λ, respectively. It then performs a *post-order* visit of T'. For each visited node n,

```
INPUT:    $T^\lambda(N,\lambda)$ /* colored policy tree */
OUTPUT:   $T'(N',\lambda')$ /* policy tree view */

MAIN
$N'$ := $N$
$\lambda'$ := $\lambda$
let $n_\top$ be the root of $T'$
Transform($n_\top$)
return($T'$)

TRANSFORM($n$)
/* visit, in the order, the children of $n$ */
for each $n_i \in children(n)$ do Transform($n_i$)
red_children := $\{n_i \in children(n)$: $\lambda'(n_i)$=red$\}$
/* prune rules */
if $\forall n_i \in children(n)$, $children(n_i)=\emptyset$ then /* internal nodes whose children are leaf nodes */
  if $\lambda'(n)$=red and $label(n)$ is a predicate operator then /* red predicate */
    remove red_children from $children(n)$
    $N'$ := $N' \setminus$ red_children
  if $children(n)$=red_children then /* red children */
    $children(n)$ := $\emptyset$
    $N'$ := $N' \setminus$ red_children
    $\lambda'(n)$ := red
/* collapse rule */
if $\lambda'(n)$=red then /* internal red nodes */
  for each $n_i \in$ red_children s.t. $children(n_i) \neq \emptyset$ then /* non-leaf red children of $n$ */
      replace $n_i$ with $children(n_i)$ in $children(n)$
      $N'$ := $N' \setminus \{n_i\}$
/* hide label rules */
if $\lambda'(n)$=red $\vee$ $\lambda'(n)$=yellow then $label(n)$ := _ /* red and yellow nodes */
```

Fig. 4. Algorithm that transforms a colored policy tree into a policy tree view

function **Transform** is recursively called on the children of n and the transformation rules in Figure 3 are applied to n as stated above. The computational cost of the transformation algorithm is $O(|N|)$ since it performs a post-order visit of T^λ, and the operations executed for each visited node have constant cost.

Example 1. Consider the colored policy tree in Figure 5(a). The algorithm in Figure 4 transforms the tree as follows. Condition (`c.type`=*IdCard*) is preserved since it is composed of green nodes only. In the subtree representing condition (`age`>*18*), the label of the yellow node *18* is removed (the structure of the subtree is instead preserved, since both node `age` and node > are green). Note that the node representing value *18* is transformed into a circle node, since we hide the fact that the original node represents a value. The subtree representing condition (`age`<*34*) is pruned, since it contains only red nodes. The visit then proceeds with red node $\wedge$ that cannot be collapsed with the red leaf node resulting from the pruning of subtree representing (`age`<*34*). Finally, the root node is visited and the collapse rule is applied, since the root has a red non-leaf child. Figure 5(b) illustrates the resulting policy view, where the red leaf represents a suppressed condition.

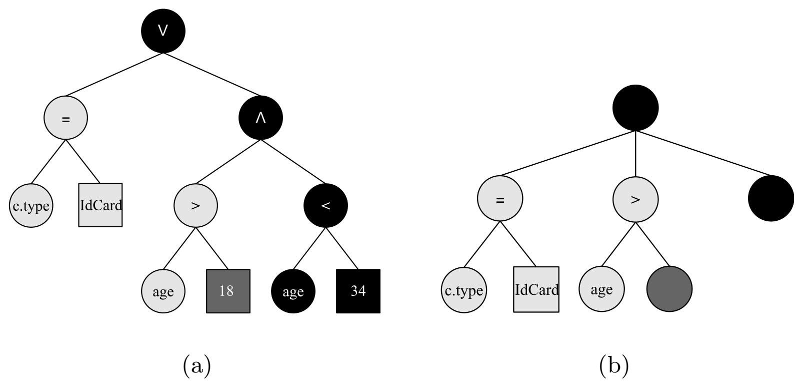

Fig. 5. An example of colored policy tree (a) and corresponding policy tree view (b)

5 Properties of the Client Policy View

Given a policy tree view T' over a colored policy tree T^λ, we identify different properties that characterize T'. These properties are useful for the server to decide whether the policy view is meaningful for the client or the application of the disclosure policies results in a view that does not represent the original policy in a “fair way”. For instance, consider the colored policy tree T^λ in Figure 5(a) and suppose that the only information visible in a policy tree view over T^λ is node `age`. In this case, a client receiving this policy tree view can only decide to release her `age`. The server can use this information to evaluate the two conditions on attribute `age` and permit or deny the access depending on whether these conditions are satisfied or not. The policy view is then meaningful for the client since it requests an attribute that is sufficient for evaluating the policy. The notions of *minimum disclosure collection* of a colored policy tree T^λ and of a policy view T' are introduced to formally capture the concept of information that permits policy evaluation. In the definition of the minimum disclosure collection,[5] conditions on attribute `type` of credentials have to be treated differently than conditions on other attributes. As already discussed in Section 3, these conditions are visible, or invisible, in their entirety. We then consider such conditions as atomic complex attributes of the policy whose label is the condition itself. In the following, when clear from the context, we will use the term attribute to refer either to an attribute `a` or to a complex attribute (c.`type` p *value*). The minimum disclosure collection can now be formally defined as follows.

Definition 3 (Minimum disclosure collection - T^λ). *Let $T^\lambda(N,\lambda)$ be a colored policy tree. The* minimum disclosure collection *$\mathcal{M}$ of T^λ is a set*

[5] A simple method consists in transforming T^λ into an equivalent tree representing the policy in disjunctive normal form; each clause of the policy corresponds to a set that contains the attributes appearing in the clause itself.

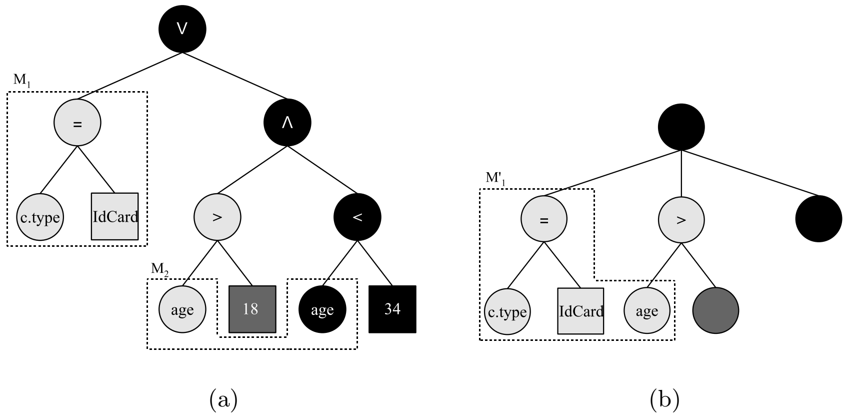

Fig. 6. An example of minimum disclosure collection of a colored policy tree (a) and corresponding policy tree view (b)

$\{M_1, \ldots, M_n\}$ *of sets, where each* $M_i, i = 1, \ldots, n$, *contains the labels of attributes whose knowledge permits the evaluation of the policy represented by* T^λ.

For instance, the policy represented by the colored policy tree in Figure 5(a) can be evaluated if the client releases either a credential of type *IdCard* or her attribute `age`. As illustrated in Figure 6(a), the corresponding minimum disclosure collection is then $\mathcal{M}$=$\{M_1$,$M_2\}$, with M_1=$\{$(`c.type` = *IdCard*)$\}$ and M_2=$\{$`age`$\}$.

The definition of the minimum disclosure collection of a policy view T' is complicated by the fact that T' can include red and/or yellow nodes, thus making difficult the determination of the attributes needed for policy evaluation. As an example, consider the policy tree view in Figure 5(b). Since the root node is red, a client does not know how the two conditions involving *c*.`type` and `age` are combined in the policy. In this case, the release of a subset of the information mentioned in the policy may not be sufficient for evaluating the corresponding policy. A client can then take a safe approach by assuming that the attributes appearing as descendants of red/yellow nodes are always all needed for policy evaluation. Intuitively, this is equivalent to say that red/yellow nodes are considered as $\wedge$ nodes. By taking into account this observation, the minimum disclosure collection of a policy tree view T' can be defined as follows.

Definition 4 (Minimum disclosure collection - T'). *Let* $T'(N',\lambda')$ *be a policy tree view, where internal red/yellow nodes are considered as* $\wedge$ *nodes. The* minimum disclosure collection $\mathcal{M}'$ *of* T' *is a set* $\{M'_1, \ldots, M'_n\}$ *of sets, where each* $M'_i, i = 1, \ldots, n$, *contains the labels of attributes whose knowledge, according to the client view, permits the evaluation of the policy represented by* T'.

For instance, the minimum disclosure collection of the policy tree view in Figure 6(b), is $\mathcal{M}'$=$\{M'_1\}$, with M'_1=$\{$(`c.type` = *IdCard*),`age`$\}$, which is obtained by considering the red root node as a $\wedge$ node; the red leaf node indicates that a condition of the original policy tree has been removed and can be ignored for the computation of $\mathcal{M}'$.

We are now ready to define the first property characterizing a policy tree view T', which we call *fair policy view*. Intuitively, a policy tree view T' is a fair policy view over T^λ if $\mathcal{M}'$ is a subset of $\mathcal{M}$. According to Definitions 3 and 4, this means that T' represents a subset of the different possible combinations of attributes and/or credentials that the client can release for policy evaluation. The following *cover* relationship is introduced for comparing the sets in the minimum disclosure collections of T^λ and T'.

Definition 5 (Cover). *Let T^λ be a colored policy tree, T' be a policy tree view over T^λ, and $\mathcal{M}$ and $\mathcal{M}'$ be their minimum disclosure collections. We say that $M'\in\mathcal{M}'$* covers *$M\in\mathcal{M}$, denoted $M\sqsubseteq M'$, iff the following conditions hold:*

1. *$\forall l\in M$ such that l is a certified attribute, then $l\in M'$;*
2. *$\forall l\in M$ such that l is a declared attribute, then $l\in M'$ or $c.l\in M'$;*
3. *$\forall b_i\in M$ such that b_i is a complex attribute, then $b_i\in M'$.*

For instance, with respect to the minimum disclosure collections in Figures 6(a) and 6(b), it is easy to see that $M_1 \sqsubseteq M'_1$ and $M_2 \sqsubseteq M'_1$. The notion of fair policy view can be now defined as follows.

Definition 6 (Fair policy view). *Let T^λ be a colored policy tree, T' be a policy tree view over T^λ, and $\mathcal{M}$ and $\mathcal{M}'$ be their minimum disclosure collections. T' is a* fair policy view *over T^λ if $\forall M'\in\mathcal{M}'$, $\exists M\in\mathcal{M}$ s.t. $M\sqsubseteq M'$ and $M'\sqsubseteq M$.*

In addition to fair, a policy tree view T' can also be characterized as *not-fair* for at least one set in $\mathcal{M}'$, *over-requesting* for all sets in $\mathcal{M}'$, or *pre-evaluable*. The evaluation of these properties of a policy view is useful to the server for deciding whether the disclosure policy specifications result in a view that can be communicated to a client. A not-fair policy view means that the policy tree view has been obfuscated by removing too much information of the original colored policy tree. There is then at least one set M' in the minimum disclosure collection $\mathcal{M}'$ of T' that does not cover any set in the minimum disclosure collection $\mathcal{M}$ of T^λ. In this case, if the client releases the information in the set M', the policy cannot be evaluated, since some information is missing. An over-requesting policy view means that the policy tree view requires more information than the minimum information needed for evaluating the corresponding policy. This happens, for example, when the label of a disjunction node has been obfuscated by coloring it in red/yellow. In this case, according to Definition 4, the red/yellow node is considered as a conjunction and then all the descendant attributes appear in a single set M'. Although more information than necessary is released, a client may still have the possibility to gain access, if the released information satisfies the corresponding policy. For instance, the policy tree view in Figure 5(b) is over-requesting since $\mathcal{M}'$=$\{M'_1\}$, and M'_1=$\{$(`c.type` = *IdCard*),`age`$\}$ includes both a condition on `type` and attribute `age`, even if only one of them would be

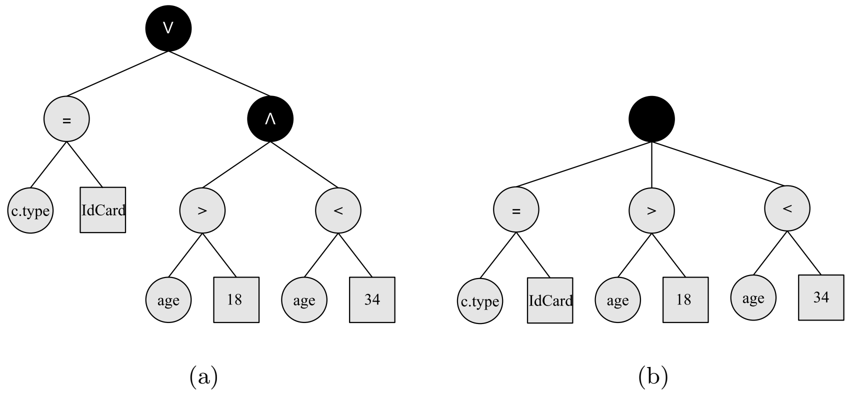

Fig. 7. An example of pre-evaluable policy tree view

sufficient for evaluating the corresponding policy. It is also important to note that even if T' is a fair or an over-requesting policy view over T^λ, a client does not have any guarantee that after releasing the information required by the policy view the access will be permitted. This guarantee holds only when the client releases a set of attributes/credentials involved in conditions that can all be evaluated at the client side and that form a set in the minimum disclosure collection of T'. In this case, if such conditions evaluate to true at the client side, they will evaluate to true at the server side. If the client can evaluate at her side all the conditions that form a set in the minimum disclosure collection of T', we say that T' is a *pre-evaluable policy view*, as formally defined in the following.

Definition 7 (Pre-evaluable policy view). *Let T^λ be a colored policy tree, T' be a policy tree view over T^λ, and $\mathcal{M}$ and $\mathcal{M}'$ be their minimum disclosure collections. T' is a* pre-evaluable policy view *if T' is a fair or an over-requesting policy view over T^λ and there is at least one $M \in \mathcal{M}$ such that all the values and predicates in the conditions involving the attributes in M correspond to green nodes in T'.*

For instance, the policy tree view in Figure 7(b) is an over-requesting, pre-evaluable representation of the colored policy tree in Figure 7(a).

6 Certificate Ontologies and Provable Conditions

In this section we briefly comment on the possibility of including certificate ontologies and provable conditions in our approach.

Certificate ontologies. Each credential in the client portfolio has a `type` that univocally determines all the other attributes appearing in the credential (i.e., all the credentials of the same type certify the same set of attributes). Credential types and their structure are known to both clients and servers and are organized through an *ontology*, which is shared among them. Each credential type in the ontology inherits all the attributes of its ancestors. The request for a credential of a given type t implicitly requires also all the attributes characterizing credentials of type t, if not differently specified. For instance, if credentials of type *IdCard* are characterized by attributes `name`, `birth_date`, and `address`, condition (c.`type`=*IdCard*) in the policy implicitly requires the disclosure of attributes `name`, `birth_date`, and `address`. We use notation $t \triangleright \texttt{a}$ to denote that credentials of type t certify attribute `a`. With reference to our example, *IdCard* $\triangleright$ `name`, *IdCard* $\triangleright$ `birth_date`, and *IdCard* $\triangleright$ `address`. The definition of cover relationship (Definition 5) can be easily revised to take into consideration the above implication between a credential type and the attributes that it certifies. A certified attribute l=c.`a` in M can be covered by the presence in M' of either l or basic condition (c.`type`=t), where $t \triangleright \texttt{a}$. Analogously, a declared attribute l=`a` in M can be covered by the presence in M' of either l, $c.l$, or basic condition (c.`type`=t), where $t \triangleright \texttt{a}$. As an example, the presence of attribute c_1.`name` in M can be covered by the presence of basic condition (c_1.`type`=*IdCard*) in M'.

Provable conditions. Modern technologies (e.g., U-Prove and Idemix [7]) permit a client to prove that her attributes satisfy a given condition, without revealing to the server any information about the attributes on which the condition is evaluated. As an example, the client can provide the server with a proof that (`age`>*18*) instead of releasing her age. Our model can be easily extended to handle this possibility. Conditions that can be proved by the client are represented in the policy tree by marking *provable* the subtree representing the condition itself. Clearly, a provable condition can only be a green subtree, since the client cannot build a proof for a condition that she does not know. Therefore, similarly to what done for conditions on credential types, provable conditions are treated as atomic elements for the computation of $\mathcal{M}$ and $\mathcal{M}'$ and are released to the client unchanged. The client who receives a request including a provable condition can decide to release either a proof, or the attributes necessary to evaluate the condition at the server side. Figure 8 illustrates an example of a colored policy tree and of the corresponding policy tree view where the double circled nodes denote the condition that can be proved by the client (i.e., (`age`>*18*)$\wedge$(`age`<*34*)). In this example: $\mathcal{M}$=$\{M_1$,$M_2\}$, with M_1=$\{$(c.`type` = *IdCard*)$\}$ and M_2=$\{$(`age`>*18*)$\wedge$(`age`<*34*)$\}$; and $\mathcal{M}'$=$\{M'_1\}$, with M'_1=$\{$(`age`>*18*)$\wedge$(`age`<*34*)$\}$. Since $M_2 \sqsubseteq M'_1$, T' is a fair and pre-evaluable policy view over the colored policy tree.

7 Related Work

The work in [3] first introduced the idea of providing fine-grained disclosure of server policy within a proposal extending XACML to support credentials, dialog management as well as more expressive conditions within policies. However this proposal considers only a limited set of policy disclosure regulations.

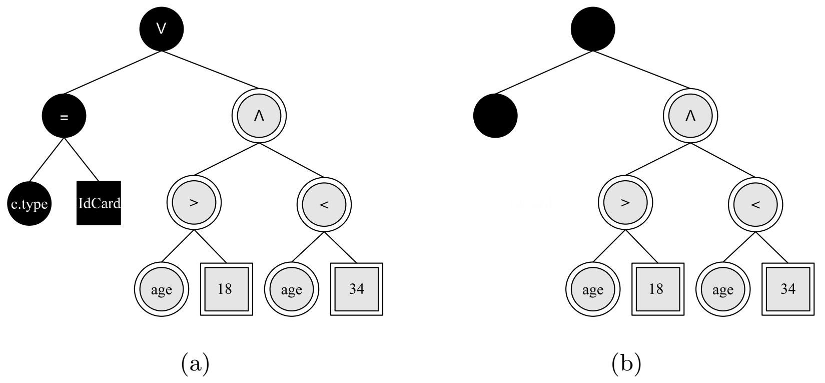

Fig. 8. An example of colored policy tree with a provable subtree (a) and the corresponding policy tree view (b)

Other related work comprises proposals aiming at protecting the client portfolio and information (e.g., [1, 2, 8, 11, 12]) and proposals enforcing trust negotiation between client and server (e.g., [6, 13–15]). In particular, [15] proposes a graph-based model for protecting the server policies released during gradual trust establishment. Each node n in the graph represents a policy that can be disclosed only if all the nodes along the path from the node representing the client request to n have been satisfied. This solution does not permit the selective disclosure of policies, but only to increase/decrease the protection of each node by moving it in the graph. The PRovisional TrUst NEgotiation (PROTUNE) [6] framework partially overcomes this issue, by identifying *private* conditions that cannot be disclosed to the client. These conditions, however, cannot be simply removed from the policy, since the client must be aware that, if a condition has been obfuscated, her request could be denied although the portion of the policy released is satisfied by the disclosed credentials. The approach illustrated in this paper is complementary to these solutions, since it can be applied at each step of the negotiation process to further limit the disclosure of the server policy. Also, our model permits to specify disclosure restrictions at a finer granularity level.

The work in [10] proposes a solution based on Identity Based Encryption that permits client-server interaction, while disclosing neither the client portfolio nor the server policy. The goal of our work is different since we aim at supporting selective disclosure of the server policy.

8 Conclusions

We presented a flexible and expressive approach for allowing the server to regulate how access control policies on its services should be communicated to clients requesting access. Our approach provides fine granularity in the specification of disclosure policies, thus capturing different ways in which the server may wish to communicate its policy. Also, our solution considering generic access control policies can be applied to different existing access control solutions, nicely complementing them with policy disclosure functionalities. To demonstrate the feasibility of the approach, we are currently implementing it within an extension of the XACML language for the support of credentials.

Acknowledgments

This work was supported in part by the EU within the 7FP project "PrimeLife" under grant agreement 216483 and by the Italian Ministry of Research within the PRIN 2008 project "PEPPER" (2008SY2PH4).

References

1. Ardagna, C., De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Samarati, P.: Minimizing disclosure of private information in credential-based interactions: A graph-based approach. In: Proc. of PASSAT 2010. Minneapolis, USA (Aug 2010)
2. Ardagna, C., De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Samarati, P.: Supporting privacy preferences in credential-based interactions. In: Proc. of WPES 2010. Chicago, USA (Oct 2010)
3. Ardagna, C., De Capitani di Vimercati, S., Paraboschi, S., Pedrini, E., Samarati, P., Verdicchio, M.: Expressive and deployable access control in open Web service applications. IEEE TSC (2010, to appear)
4. Bonatti, P., Samarati, P.: A unified framework for regulating access and information release on the Web. JCS 10(3), 241–272 (2002)
5. Bonatti, P., Samarati, P.: Logics for authorizations and security. In: Chomicki, J., et al. (eds.) Logics for Emerging Applications of Databases. Springer-Verlag (2003)
6. Bonatti, P., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: Proc. of POLICY 2005. Stockholm, Sweden (June 2005)
7. Camenisch, J., Lysyanskaya, A.: An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In: Proc. of EUROCRYPT 2001. Innsbruck, Austria (May 2001)
8. Cimato, S., Gamassi, M., Piuri, V., Sassi, R., Scotti, F.: Privacy-aware biometrics: Design and implementation of a multimodal verification system. In: Proc. of ACSAC 2008. Anaheim, USA (Dec 2008)
9. eXtensible Access Control Markup Language (XACML) v2.0 (Feb 2005), http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf
10. Frikken, K., Atallah, M., Li, J.: Attribute-based access control with hidden policies and hidden credentials. IEEE TC 55(10), 1259–1270 (2006)
11. Gamassi, M., Lazzaroni, M., Misino, M., Piuri, V., Sana, D., Scotti, F.: Accuracy and performance of biometric systems. In: Proc. of IMTC 2004. Como, Italy (2004)
12. Gamassi, M., Piuri, V., Sana, D., Scotti, F.: Robust fingerprint detection for access control. In: Proc. of RoboCare Workshop 2005. Rome, Italy (May 2005)
13. Irwin, K., Yu, T.: Preventing attribute information leakage in automated trust negotiation. In: Proc. of CCS 2005. Alexandria, USA (Nov 2005)
14. Lee, A., Winslett, M., Basney, J., Welch, V.: The Traust authorization service. ACM TISSEC 11(1), 1–3 (2008)
15. Yu, T., Winslett, M.: A unified scheme for resource protection in automated trust negotiation. In: Proc. of IEEE S&P 2003. Berkeley, USA (May 2003)